To determine the order of growth of the running time of each of the four Java functions given (each of which returns a string of length n whose characters are all x), we need to analyze the time complexity of each method. Let's evaluate each function:

Method 1

```java
public static String method1(int n) {
    if (n == 0) return "";
    String temp = method1(n / 2);
    if (n % 2 == 0) return temp + temp;
    else return temp + temp + "x";
}
```

- **Analysis**: This method recursively divides `n` by 2. For each recursive call, it concatenates two strings of length `n/2` or `n/2 + 1`. The concatenation takes time proportional to the length of the resulting string.

Breaking Down the Method

1. **Base Case**:
   - If `n == 0`, the method returns an empty string, which is a constant-time operation: $T(0) = O(1)$.
2. **Recursive Call**:
   - The method calls itself with `n / 2` (integer division). This divides the problem into smaller subproblems.
   - Let's denote the time complexity of this recursive call as $T(n/2)$.
3. **Concatenation**:
   - After the recursive call, we have a string `temp` of length approximately $n/2$.
   - If `n` is even, `temp + temp` concatenates two strings of length $n/2$, resulting in a string of length n. The time complexity of this concatenation is proportional to the length of the resulting string, which is $O(n)$.
   - If `n` is odd, `temp + temp + "x"` concatenates two strings of length $n/2$ and an additional character, resulting in a string of length n. Again, the time complexity of this concatenation is $O(n)$.

Formulating the Recurrence Relation

Considering the above steps, the recurrence relation for the running time $T(n)$ can be described as follows:

$$T(n) = T(n/2) + O(n)$$

Combining the Terms

Combining these, we get the recurrence relation:

$$T(n) = 2T(n/2) + O(n)$$

This relation indicates that at each step, we solve one subproblem of size $n/2$ and then perform an $O(n)$ concatenation operation. Notice that the factor of 2 in the recursion arises because we concatenate `temp` with itself, effectively doubling the work done in the concatenation step.

- **Time complexity**: Solving this using the Master Theorem, we get: $T(n) = O(n \log n)$

The Master Theorem provides a straightforward way to determine the asymptotic behavior of recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where:

- $a \geq 1$ is the number of subproblems in the recurrence,
- $\frac{n}{b}$ is the size of each subproblem (with $b > 1$),
- $f(n)$ is the cost outside the recursive calls, often representing the time to divide the problem and combine the results of the subproblems.

The Master Theorem considers three cases to determine the order of growth of the recurrence:

1. **Case 1**: If $f(n) = O(n^c)$ where $c < \log_b a$, then: $T(n) = O(n^{\log_b a})$
2. **Case 2**: If $f(n) = O(n^c)$ where $c = \log_b a$, then: $T(n) = O(n^c \log n)$
3. **Case 3**: If $f(n) = O(n^c)$ where $c > \log_b a$, then: $T(n) = O(f(n))$

Explanation of the Cases

- **Case 1**: The work done outside the recursive calls, $f(n)$, is polynomially smaller than the work done inside the recursive calls. The overall time complexity is dominated by the recursive calls.
- **Case 2**: The work done outside the recursive calls, $f(n)$, is asymptotically the same as the work done inside the recursive calls. The overall time complexity is the product of the two, with a logarithmic factor.
- **Case 3**: The work done outside the recursive calls, $f(n)$, is polynomially larger than the work done inside the recursive calls. The overall time complexity is dominated by $f(n)$.

Applying the Master Theorem

The recurrence relation for both method 1 and 3 is: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

Here:

- $a = 2$
- $b = 2$
- $f(n) = O(n)$

We need to compare $f(n)$ with $n^{\log_b a}$:

- $\log_b a = \log_2 2 = 1$
- So $n^{\log_b a} = n^1 = n$

Since $f(n) = O(n)$ matches $n^{\log_b a}$, we are in Case 2: $T(n) = O(n \log n)$

Thus, for both Method 1 and Method 3, the time complexity is $O(n \log n)$.

Method 2

```
public static String method2(int n) {
    String s = "";
    for (int i = 0; i < n; i++)
        s = s + "x";
    return s;
}
```

- **Analysis**: This method concatenates a single character `x` to the string `s` in each iteration of the loop. Each concatenation takes time proportional to the length of the resulting string.
- **Time complexity**: The total time for all concatenations is: $T(n) = 1 + 2 + 3 + \ldots + n = O(n^2)$

The key to understanding the time complexity of this method lies in the way string concatenation works in Java. Strings in Java are immutable, meaning that every time you concatenate two strings, a new string is created, and the contents of both original strings are copied into this new string.

Let's break down the steps involved in each iteration of the loop:

1. **Initialization**: `s` is initially an empty string (`""`).
2. **First iteration (i = 0)**:
   - `s = s + "x"`
   - The length of `s` is 1 after this operation.
   - This operation takes $O(1)$ time.
3. **Second iteration (i = 1)**:
   - `s = s + "x"`
   - The length of `s` is 2 after this operation.

- This operation takes $O(2)$ time because it involves copying the 1-character string and appending another character.

4. **Third iteration (i = 2)**:
   - `s = s + "x"`
   - The length of `s` is 3 after this operation.
   - This operation takes $O(3)$ time because it involves copying the 2-character string and appending another character.
5. **And so on...**

In general, during the i-th iteration, the length of `s` is i, and the concatenation operation takes $O(i)$ time.

Summing Up the Costs

The total time $T(n)$ for executing the loop is the sum of the time taken for each iteration:

$$T(n) = \sum_{i=0}^{n-1} O(i) = O(0) + O(1) + O(2) + \ldots + O(n-1)$$

This sum is equivalent to the sum of the first $n-1$ integers:

$$T(n) = \sum_{i=0}^{n-1} i = \frac{(n-1)n}{2}$$

The dominant term here is $\frac{n^2}{2}$, which simplifies to $O(n^2)$ in Big-O notation.

Method 3

```
public static String method3(int n) {
    if (n == 0) return "";
    if (n == 1) return "x";
    return method3(n/2) + method3(n - n/2);
}
```

- **Analysis**: This method recursively divides `n` into two parts: `n/2` and `n - n/2`. It then concatenates the results of these two parts.
- **Time complexity**: The recurrence relation for the running time $T(n)$ is: $T(n) = 2T(n/2) + O(n)$ This is the same as the recurrence relation for Method 1. Solving this using the Master Theorem, we get: $T(n) = O(n \log n)$

Method 4

```
public static String method4(int n) {
    char[] temp = new char[n];
    for (int i = 0; i < n; i++)
        temp[i] = 'x';
    return new String(temp);
}
```

- **Analysis**: This method fills an array of size `n` with the character `x` in a single loop, then creates a string from this array.
- **Time complexity**: Filling the array takes $O(n)$ time, and creating the string from the array also takes $O(n)$ time. $T(n) = O(n)$

Summary

- **Method 1**: $O(n \log n)$
- **Method 2**: $O(n^2)$
- **Method 3**: $O(n \log n)$
- **Method 4**: $O(n)$